MIXED STACK-BASED RISC PROCESSOR

CROSS-REFERENCE TO RELATED APPLICATIONS

[0001] This application claims priority to U.S. Provisional Application Serial No. 60/400,391 titled "JSM Protection," filed July 31, 2002, incorporated herein by reference. This application also claims priority to EPO Application No. 03291916.9, filed July 30, 2003 and entitled "Mixed Stack-Based RISC Processor," incorporated herein by reference. This application also may contain subject matter that may relate to the following commonly assigned co-pending applications incorporated herein by reference: "System And Method To Automatically Stack And Unstack Java Local Variables," Serial No. _____, filed July 31, 2003, Attorney Docket No. TI-35422 (1962-05401); "Memory Management Of Local Variables," Serial No. _____, filed July 31, 2003, Attorney Docket No. TI-35423 (1962-05402); "Memory Management Of Local Variables Upon A Change Of Context," Serial No. _____, filed July 31, 2003, Attorney Docket No. TI-35424 (1962-05403); "A Processor With A Split Stack," Serial No. _____, filed July 31, 2003, Attorney Docket No. TI-35425(1962-05404); "Using IMPDEP2 For System Commands Related To Java Accelerator Hardware," Serial No. _____, filed July 31, 2003, Attorney Docket No. TI-35426 (1962-05405); "Test With Immediate And Skip Processor Instruction," Serial No. _____, filed July 31, 2003, Attorney Docket No. TI-35427 (1962-05406); "Test And Skip Processor Instruction Having At Least One Register Operand," Serial No. _____, filed July 31, 2003, Attorney Docket No. TI-35248 (1962-05407); "Synchronizing Stack Storage," Serial No. _____, filed July 31, 2003, Attorney Docket No. TI-35429 (1962-05408); "Methods And Apparatuses For Managing Memory," Serial No. _____, filed July 31, 2003, Attorney Docket

No. TI-35430 (1962-05409); "Write Back Policy For Memory," Serial No. _____, filed July 31, 2003, Attorney Docket No. TI-35431 (1962-05410); "Methods And Apparatuses For Managing Memory," Serial No. _____, filed July 31, 2003, Attorney Docket No. TI-35432 (1962-05411); "Processor That Accommodates Multiple Instruction Sets And Multiple Decode Modes," Serial No. _____, filed July 31, 2003, Attorney Docket No. TI-35434 (1962-05413); "System To Dispatch Several Instructions On Available Hardware Resources," Serial No. _____, filed July 31, 2003, Attorney Docket No. TI-35444 (1962-05414); "Micro-Sequence Execution In A Processor," Serial No. _____, filed July 31, 2003, Attorney Docket No. TI-35445 (1962-05415); "Program Counter Adjustment Based On The Detection Of An Instruction Prefix," Serial No. _____, filed July 31, 2003, Attorney Docket No. TI-35452 (1962-05416); "Reformat Logic To Translate Between A Virtual Address And A Compressed Physical Address," Serial No. _____, filed July 31, 2003, Attorney Docket No. TI-35460 (1962-05417); "Synchronization Of Processor States," Serial No. _____, filed July 31, 2003, Attorney Docket No. TI-35461 (1962-05418); "Conditional Garbage Based On Monitoring To Improve Real Time Performance," Serial No. _____, filed July 31, 2003, Attorney Docket No. TI-35485 (1962-05419); "Inter-Processor Control," Serial No. _____, filed July 31, 2003, Attorney Docket No. TI-35486 (1962-05420); "Cache Coherency In A Multi-Processor System," Serial No. _____, filed July 31, 2003, Attorney Docket No. TI-35637 (1962-05421); "Concurrent Task Execution In A Multi-Processor, Single Operating System Environment," Serial No. _____, filed July 31, 2003, Attorney Docket No. TI-35638 (1962-05422); and "A Multi-Processor Computing System Having A Java Stack Machine And A RISC-Based Processor," Serial No. _____, filed July 31, 2003, Attorney Docket No. TI-35710 (1962-05423).

BACKGROUND OF THE INVENTION

Technical Field of the Invention

[0002] The present invention relates generally to processors and more particularly to a processor capable of executing a stack-based instruction set and a non-stack based instruction set.

Background Information

[0003] Many types of electronic devices are battery operated and thus preferably consume as little power as possible. An example is a cellular telephone. Further, it may be desirable to implement various types of multimedia functionality in an electronic device such as a cell phone. Examples of multimedia functionality may include, without limitation, games, audio decoders, digital cameras, etc. It is thus desirable to implement such functionality in an electronic device in a way that, all else being equal, is fast, consumes as little power as possible and requires as little memory as possible. Improvements in this area are desirable.

BRIEF SUMMARY

[0004] As disclosed herein, a processor (e.g., a co-processor) executes a stack-based instruction set and another instruction set in a way that accelerates the execution of the stack-based instruction set, although code acceleration is not required under the scope of this disclosure. In accordance with at least some embodiments of the invention, the processor may comprise a multi-entry stack usable in at least a stack-based instruction set, logic coupled to and managing the stack, and a plurality of registers coupled to the logic and addressable through a second instruction set that provides register-based and memory-based operations.

[0005] Other embodiments may include a system (e.g., a cellular telephone) that includes a main processor unit coupled to a co-processor. The co-processor may be configured to execute stack-

based instructions from a first instruction set and instructions from a second instruction set that provides memory-based and register-based operations.

[0006] The processor described herein may include a multi-entry stack and registers at least some of which store the address of the top of the stack and the data value at the top of the stack. This multi-entry stack is generally fabricated in the core of the processor and represents the top *n* (e.g., eight) entries of a larger stack implemented in memory. These and other features are described herein.

NOTATION AND NOMENCLATURE

[0007] Certain terms are used throughout the following description and claims to refer to particular system components. As one skilled in the art will appreciate, semiconductor companies may refer to a component by different names. This document does not intend to distinguish between components that differ in name but not function. In the following discussion and in the claims, the terms "including" and "comprising" are used in an open-ended fashion, and thus should be interpreted to mean "including, but not limited to...". Also, the term "couple" or "couples" is intended to mean either an indirect or direct connection. Thus, if a first device couples to a second device, that connection may be through a direct connection, or through an indirect connection via other devices and connections.

BRIEF DESCRIPTION OF THE DRAWINGS

[0008] For a more detailed description of the preferred embodiments of the present invention, reference will now be made to the accompanying drawings, wherein:

[0009] Figure 1 shows a diagram of a system in accordance with preferred embodiments of the invention and including a Java Stack Machine ("JSM") and a Main Processor Unit ("MPU");

[0010] Figure 2 shows a block diagram of the JSM of Figure 1 in accordance with preferred embodiments of the invention;

[0011] Figure 3 shows various registers used in the JSM of Figures 1 and 2;

[0012] Figure 4 shows an embodiment useful to illustrate various addressing modes in accordance with the preferred embodiments;

[0013] Figures 5A-L show exemplary formats of various instructions executed by the JSM of Figures 1 and 2; and

[0014] Figure 6 depicts an exemplary embodiment of the system described herein.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS

[0015] The following discussion is directed to various embodiments of the invention. Although one or more of these embodiments may be preferred, the embodiments disclosed should not be interpreted, or otherwise used, as limiting the scope of the disclosure, including the claims, unless otherwise specified. In addition, one skilled in the art will understand that the following description has broad application, and the discussion of any embodiment is meant only to be exemplary of that embodiment, and not intended to intimate that the scope of the disclosure, including the claims, is limited to that embodiment.

[0016] The subject matter disclosed herein is directed to a programmable electronic device such as a processor. The processor described herein is particularly suited for executing Java™ Bytecodes or comparable, code. As is well known, Java is particularly suited for embedded applications. Java is a relatively "dense" language meaning that on average each instruction may perform a large number of functions compared to various other programming languages. The dense nature of Java

is of particular benefit for portable, battery-operated devices that preferably include as little memory as possible to save space and power. The reason, however, for executing Java code is not material to this disclosure or the claims which follow. The processor described herein may be used in a wide variety of electronic systems. By way of example and without limitation, the Java-executing processor described herein may be used in a portable, battery-operated cell phone. Further, the processor advantageously includes one or more features that permit the execution of the Java code to be accelerated.

[0017] Referring now to Figure 1, a system 100 is shown in accordance with a preferred embodiment of the invention. As shown, the system includes at least two processors 102 and 104. Processor 102 is referred to for purposes of this disclosure as a Java Stack Machine ("JSM") and processor 104 may be referred to as a Main Processor Unit ("MPU"). System 100 may also include memory 106 coupled to both the JSM 102 and MPU 104 and thus accessible by both processors. At least a portion of the memory 106 may be shared by both processors meaning that both processors may access the same shared memory locations. Further, if desired, a portion of the memory 106 may be designated as private to one processor or the other. System 100 also includes a Java Virtual Machine ("JVM") 108, compiler 110, and a display 114. The JSM 102 preferably includes an interface to one or more input/output ("I/O") devices such as a keypad to permit a user to control various aspects of the system 100. In addition, data streams may be received from the I/O space into the JSM 102 to be processed by the JSM 102. Other components (not specifically shown) may include, without limitation, a battery and an analog transceiver to permit wireless communications with other devices. As noted above, while system 100 may be representative of, or adapted to, a wide variety of electronic systems, an exemplary electronic system may comprise a battery-operated, mobile cell phone.

[0018] As is generally well known, Java code comprises a plurality of "bytecodes" 112. Bytecodes 112 may be provided to the JVM 108, compiled by compiler 110 and provided to the JSM 102 and/or MPU 104 for execution therein. In accordance with a preferred embodiment of the invention, the JSM 102 may execute at least some, and generally most, of the Java bytecodes. When appropriate, however, the JSM 102 may request the MPU 104 to execute one or more Java bytecodes not executed or executable by the JSM 102. In addition to executing Java bytecodes, the MPU 104 also may execute non-Java instructions. The MPU 104 also hosts an operating system ("O/S") (not specifically shown), which performs various functions including system memory management, the system task management that schedules the JVM 108 and most or all other native tasks running on the system, management of the display 114, receiving input from input devices, etc. Without limitation, Java code may be used to perform any one of a variety of applications including multimedia, games or web based applications in the system 100, while non-Java code, which may comprise the O/S and other native applications, may still run on the system on the MPU 104.

[0019] The JVM 108 generally comprises a combination of software and hardware. The software may include the compiler 110 and the hardware may include the JSM 102. The JVM may include a class loader, bytecode verifier, garbage collector, and a bytecode interpreter loop to interpret the bytecodes that are not executed on the JSM processor 102.

[0020] In accordance with preferred embodiments of the invention, the JSM 102 may execute at least two instruction sets. One instruction set may comprise standard Java bytecodes. As is well-known, Java is a stack-based programming language in which instructions generally target a stack. For example, an integer add ("IADD") Java instruction pops two integers off the top of the stack, adds them together, and pushes the sum back on the stack. As will be explained in more detail

below, the JSM 102 comprises a stack-based architecture with various features that accelerate the execution of stack-based Java code.

[0021] Another instruction set executed by the JSM 102 may include instructions other than standard Java instructions. In accordance with at least some embodiments of the invention, such other instruction set may include register-based and memory-based operations to be performed. This other instruction set generally complements the Java instruction set and, accordingly, may be referred to as a complementary instruction set architecture ("C-ISA"). By complementary, it is meant that the execution of more complex Java Bytecodes may be substituted by a "micro-sequence" comprising one or more C-ISA instructions that permit address calculation to readily "walk through" the JVM data structures. A micro-sequence also may include one or more Bytecode instructions. The execution of Java may be made more efficient and run faster by replacing some sequences of Bytecodes by preferably shorter and more efficient sequences of C-ISA instructions. The two sets of instructions may be used in a complementary fashion to obtain satisfactory code density and efficiency. As such, the JSM 102 generally comprises a stack-based architecture for efficient and accelerated execution of Java bytecodes combined with a register-based architecture for executing register and memory based C-ISA instructions. Both architectures preferably are tightly combined and integrated through the C-ISA.

[0022] Figure 2 shows an exemplary block diagram of the JSM 102. As shown, the JSM includes a core 120 coupled to data storage 122 and instruction storage 130. The core may include one or more components as shown. Such components preferably include a plurality of registers 140, three address generation units ("AGUs") 142, 147, micro-translation lookaside buffers (micro-TLBs) 144, 156, a multi-entry micro-stack 146, an arithmetic logic unit ("ALU") 148, a multiplier 150, decode logic 152, and instruction fetch logic 154. In general, operands may be retrieved from data

storage 122 or from the micro-stack 146, processed by the ALU 148, while instructions may be fetched from instruction storage 130 by fetch logic 154 and decoded by decode logic 152. The address generation unit 142 may be used to calculate addresses based, at least in part on data contained in the registers 140. The AGUs 142 may calculate addresses for C-ISA instructions as will be described below. The AGUs 142 may support parallel data accesses for C-ISA instructions that perform array or other types of processing. AGU 147 couples to the micro-stack 146 and may manage overflow and underflow conditions in the micro-stack preferably in parallel. The micro-TLBs 144, 156 generally perform the function of a cache for the address translation and memory protection information bits that are preferably under the control of the operating system running on the MPU 104.

[0023] Referring now to Figure 3, the registers 140 may include 16 registers designated as R0-R15. Registers R0-R3, R5, R8-R11 and R13-R14 may be used as general purposes ("GP") registers usable for any purpose by the programmer. Other registers, and some of the GP registers, may be used for specific functions. For example, registers R4 and R12 may be used to store two program counters. Register R4 preferably is used to store the program counter ("PC") and register R12 preferably is used to store a micro-program counter ("micro-PC"). In addition to use as a GP register, register R5 may be used to store the base address of a portion of memory in which Java local variables may be stored when used by the current Java method. The top of the micro-stack 146 is reflected in registers R6 and R7. The top of the micro-stack has a matching address in external memory pointed to by register R6. The values contained in the micro-stack are the latest updated values, while their corresponding values in external memory may or may not be up to date. Register R7 provides the data value stored at the top of the micro-stack. Registers R8 and R9 may also be used to hold the address index 0 ("AI0") and address index 1 ("AI1"), as will be

explained below. Register R14 may also be used to hold the indirect register index ("IRI") as will also be explained below. Register R15 may be used for status and control of the JSM 102. As an example, one status/control bit (called the "Micro-Sequence-Active" bit) may indicate if the JSM 102 is executing a "simple" instruction or a "complex" instruction through a "micro-sequence." This bit controls in particular, which program counter is used R4 (PC) or R12 (micro-PC) to fetch the next instruction. A "simple" Bytecode instruction is generally one in which the JSM 102 may perform an immediate operation either in a single cycle (e.g., an "iadd" instruction) or in several cycles (e.g., "dup2_x2"). A "complex" Bytecode instruction is one in which several memory accesses may be required to be made within the JVM data structure for various verifications (e.g., NULL pointer, array boundaries). Because these data structure are generally JVM-dependent and thus may change from one JVM implementation to another, the software flexibility of the micro-sequence provides a mechanism for various JVM optimizations now known or later developed.

[0024] Referring again to Figure 2, as noted above, the JSM 102 is adapted to process and execute instructions from at least two instruction sets. One instruction set includes stack-based operations and the second instruction set includes register-based and memory-based operations. The stack-based instruction set may include Java bytecodes. Java bytecodes pop, unless empty, data from and push data onto the micro-stack 146. The micro-stack 146 preferably comprises the top *n* entries of a larger stack that is implemented in data storage 122. Although the value of *n* may be vary in different embodiments, in accordance with at least some embodiments, the size *n* of the micro-stack may be the top eight entries in the larger, memory-based stack. The micro-stack 146 preferably comprises a plurality of gates in the core 120 of the JSM 102. By implementing the micro-stack 146 in gates (e.g., registers) in the core 120 of the processor 102, access to the data

contained in the micro-stack 146 is generally very fast, although any particular access speed is not a limitation on this disclosure.

[0025] The second, register-based, memory-based instruction set may comprise the C-ISA instruction set introduced above. The C-ISA instruction set preferably is complementary to the Java bytecode instruction set in that the C-ISA instructions may be used to accelerate or otherwise enhance the execution of Java bytecodes.

[0026] The ALU 148 adds, subtracts, and shifts data. The multiplier 150 may be used to multiply two values together in one or more cycles. The instruction fetch logic 154 generally fetches instructions from instruction storage 130. The instructions may be decoded by decode logic 152. Because the JSM 102 is adapted to process instructions from at least two instruction sets, the decode logic 152 generally comprises at least two modes of operation, one mode for each instruction set. As such, the decode logic unit 152 may include a Java mode in which Java instructions may be decoded and a C-ISA mode in which C-ISA instructions may be decoded.

[0027] The data storage 122 generally comprises data cache ("D-cache") 124 and data random access memory ("D-RAMset") 126. Reference may be made to copending applications U.S. Serial nos. 09/591,537 filed June 9, 2000 (atty docket TI-29884), 09/591,656 filed June 9, 2000 (atty docket TI-29960), and 09/932,794 filed August 17, 2001 (atty docket TI-31351), all of which are incorporated herein by reference. The stack (excluding the micro-stack 146), arrays and non-critical data may be stored in the D-cache 124, while Java local variables, critical data and non-Java variables (e.g., C, C++) may be stored in D-RAM 126. The instruction storage 130 may comprise instruction RAM ("I-RAM") 132 and instruction cache ("I-cache") 134. The I-RAMset 132 may be used for "complex" micro-sequenced Bytecodes or micro-sequences or predetermined

sequences of code, as will be described below. The I-cache 134 may be used to store other types of Java bytecode and mixed Java/C-ISA instructions.

[0028] As noted above, the C-ISA instructions generally complement the standard Java bytecodes. For example, the compiler 110 may scan a series of Java bytes codes 112 and replace one or more of such bytecodes with an optimized code segment mixing C-ISA and bytecodes and which is capable of more efficiently performing the function(s) performed by the initial group of Java bytecodes. In at least this way, Java execution may be accelerated by the JSM 102.

[0029] As explained above, the C-ISA instruction set preferably permits register-based and memory-based operations. Memory-based operations generally require the calculation of an address in memory for the operand or the result. The AGUs 142 are used to calculate such memory references. Referring briefly to Figure 4, one or more values may be provided to an AGU 142. Such values may include any, or all, of a source address 200, an immediate value 202, and an offset address 204. Such values may be included in, or otherwise specified by, a C-ISA instruction. These values also may be implicit or explicit within the Java bytecodes. For example, "Iload_1" may generate an address from the base address of the local variable area present in register R5 plus one to access the local variable 1 within the D_RAMset. The "iload" <index> may generate an address from the base address of the local variable area present in register R5 plus the value of the "index" operand. Further, the JSM 102 preferably is capable of multiple addressing modes as explained below.

[0030] Figures 5A-5L provide formats for various embodiments of C-ISA instructions. Figures 5A-5H show 16 bit formats, while Figures 5I-L show 32 bit formats. Figures 5A-5E generally include the same format which comprises an opcode field 230, an operand field 232 and an immediate value field 234. The number of bits in each field may be varied, but in accordance with

the preferred embodiment, the opcode field 230 comprises four bits, the operand field 232 comprises four bits and the immediate value field 234 comprises eight bits.

[0031] With four bits, the opcode field 230 may encode up to 16 different, 16-bit instructions or groups of instructions. In particular, some of the four bit values may be used to extend the size of the instruction to 32-bits. Referring still to Figures 5A-5E, the immediate field 234 preferably comprises a seven-bit immediate value V and an associated sign bit S. As such, the immediate value field 234 may comprise values in the range from –127 to +128. The operand field 232 is used to store the identity of one of the registers 140. Because there are 16 registers, the operand field 232 includes four bits so that all 16 registers can be used in the instructions of Figures 5A-E.

[0032] The "load immediate" instruction 236 shown in Figure 5A causes the immediate value from field 234 to be loaded into the destination register "R_d" specified in field 232. The "add immediate" instruction 238 shown in Figure 5B causes the immediate value in field 234 to be added to the contents of the destination register "R_d" field 232 and the result stored back in R_d.

[0033] In Figure 5C, the "AND with immediate" instruction 240 causes the immediate value in field 234 to be logically AND'd with the value held within the destination register R_d. The result is stored back in R_d. The "OR with immediate" instruction 242 in Figure 5D causes the immediate value in field 234 to be logically OR'd with the value held within the destination register R_d corresponding to field 232 and the result stored back in R_d.

[0034] The "test with immediate and skip" instruction 244 in Figure 5E causes the immediate value in field 234 to be compared to the contents of the destination register R_d and a bit in the status register R15 to be written as a 0 or 1 to reflect the result of the comparison. If the immediate value does not match the contents of R_d then the instruction (not specifically shown in Figure 5E) following the "test with immediate and skip" instruction is skipped during execution, for example,

by changing the next instruction into a no operation instruction ("NOP"). If, however, the immediate value does match the contents of the target memory location, then the subsequent instruction is not skipped.

[0035] Figure 5F shows a preferred format for a 16-bit load or store instruction 246 in which the value being loaded or stored is not expressly included in the instruction itself, as was the case with the load immediate instruction 236 described above in Figure 5A. In Figure 5F, the opcode field encodes either a load opcode or a store opcode. A plurality of addressing modes are possible with the exemplary format of Figure 5F and, in accordance with the preferred embodiments of the present invention, four addressing modes are possible. Bits P 250 and I 256 may encode any one of the four addressing modes to be used in a particular instance of load/store instruction 246. In each addressing mode, two locations are used—data is retrieved from one location and stored at another. In the context of a load instruction, data is read from a "source" address in memory and loaded into a "target" register. For a store instruction, data is read from a target register "Rd" which is included in field 232 and stored in memory at a source address in accordance with any of a plurality of addressing modes as explained below. The Rd field 232 includes three bits in the embodiment of Figure 5F, and thus encodes one of eight registers (preferably registers R0-R7).

[0036] The following discussion described four addressing modes for computing the source address in accordance with the preferred embodiments of the invention. If the P bit 250 is a 0 and the I bit 256 is a 0, the source address preferably is computed by adding together a base address and an immediate value. The base address is stored in a register which is specified by the R_s field 252. The immediate value is specified in the immediate field 254 which comprises a sign bit S and an immediate bit V. As such, the immediate field 254 may include an immediate value including –

1, 0, 1, and 2. Like the R_d field 232, the R_s field 252 may include three bits and as such, may encode one of eight registers, preferably R0-R7.

[0037] If the P bit 250 is a 0 and the I bit 256 is a 1, the source memory address is the base address specified by the R_s register in field 252. Once the base address is used to determine the memory reference, the R_s register value in field 252 is recomputed by adding the current R_s register value by the amount of the immediate field 254 and storing the result back in the R_s field 252 (i.e., a post increment by an amount V).

[0038] If the P bit 250 is a 1 and the I bit 256 is a 0, the source memory address preferably is computed by adding together the memory address contained in the R_s register (field 252) and the memory address contained in a predetermined index register (e.g., register R8). Once the source address is computed, the value in the predetermined index register may be incremented by the amount of the immediate field 254, an amount which may range from –1 to +2 as explained above.

[0039] If the P bit 250 is a 1 and the I bit 256 is a 1, the source memory address preferably is computed by adding together the memory address contained in the R_s register identified in field 252 and the memory address contained in the predetermined index register R8. The sum of those two memory addresses represents the source address for the load or store instruction.

[0040] As explained above, the R_d register represents the target register for the data transfer. That register may include any register R0-R7. The R7 register is the top of the stack register. If the R_d register in the load/store instruction 246 is the R7 register, then the R6 register, which includes the stack pointer (SP), preferably is updated accordingly to reflect a change in the status of the micro-stack 146 and the value from the top of the stack (R7) that is used is consumed (i.e., removed). The inclusion of register R7 storing the top of the stack permits an efficient and powerful mechanism for transferring blocks of data to/from memory from/to the stack with a single

instruction with a repeat loop. The AGUs 142 may also be used in this context to calculate memory source and destination addresses for such data transfers.

[0041] Turning now to Figure 5G, a conditional control instruction 260 is shown which preferably causes the value held in R_d (field 232) to be compared with the value located in memory (addressing mode already described). The comparison may determine whether the R_d value is not equal, equal, greater than, greater than or equal, smaller than, or smaller than or equal to the value located in memory. Other comparison conditions may also be defined as desired. The type of comparison being made is encoded in the COND field 262 which may comprise three bits as shown. In this instruction, when R_d is equal to R_7, the stack pointer in register R_6 is updated accordingly when, for example, a value is consumed from the stack.

[0042] Figures 5H-L show various 32-bit C-ISA instructions. Turning to Figure 5H, an array processing instruction takes a first operand array from memory located at an address present in an implicit register (e.g., register R0) and a second operand array from memory located at an address present in another implicit register (e.g., R1) with all possible addressing modes defined above and performs the operation defined by the opcode field 239. Without limitation, the operation may include operations such as add, subtract, AND, OR, etc. Register R8 and immediate value V1 are associated with the operand of a first array, while register R9 and the immediate value V2 are associated with the operand of a second array. The result of the operation preferably is pushed onto the stack and thus the use of register R7 is implicit in the instruction and thus register R6 is updated by each of these instructions.

[0043] In Figure 5I, a 32-bit load/store instruction 286 is shown in which data is loaded in register Rd if a load is specified in field 230. If a store is specified, the data to be stored is specified in Rd. Four different addressing modes can be implemented in instruction 286 as described above via the

encoding of the P and I bits 250, 256. If Rd or Rs is equal to R7, then the stack pointer register R6 is updated accordingly as explained previously. The 32-bit load/store instruction 286 is generally similar to its 16-bit counterpart instruction 246 in Figure 5F, but the immediate field 288 in instruction 286 is wider (12 bits including sign bit S) than in instruction 246, permitting increased code density.

[0044] Turning now to Figure 5J, an I/O access instruction 302 is shown which preferably loads or stores data between the I/O space and the JSM register set 140. The R_d register generally is used to receive data in an I/O read and as a source register in the event of an I/O wrirte. As before, multiple addressing schemes may be specified by the P and I bits 250, 256. As before, register R6 is updated accordingly if Rd or Rs equals R7.

[0045] Turning now to Figure 5K, a register shift instruction 318 permits the data in R_d to be shifted left or right. The D bit 320 encodes the direction of the shift (left or right). The Ir bit 322 specifies whether the shift value comes from an immediate value V 324 or from Rs 326. If Rd equals R7, then top of stack pointer R6 is updated as before. The T bit indicates the type of shift operation (e.g., arithmetic or logical).

[0046] Turning now to Figure 5L, a register arithmetic instruction 334 preferably takes two operands from R_{s1} and R_{s2}, performs an operation (e.g., add, subtract, AND, OR, XOR, etc.) defined by the opcode field on these values, and stores the result into R_d. Register R6 is updated in the event R7 is loaded into R_d or either of R_{s1} or R_{s2} is equal to R7.

[0047] As noted previously, system 100 may be implemented as a mobile cell phone such as that shown in Figure 6. As shown, a mobile communication device includes an integrated keypad 412 and display 414. The JSM processor 102 and MPU processor 104 and other components may be

included in electronics package 410 connected to the keypad 412, display 414, and radio frequency ("RF") circuitry 416. The RF circuitry 416 may be connected to an antenna 418.

[0048] While the preferred embodiments of the present invention have been shown and described, modifications thereof can be made by one skilled in the art without departing from the spirit and teachings of the invention. The embodiments described herein are exemplary only, and are not intended to be limiting. Many variations and modifications of the invention disclosed herein are possible and are within the scope of the invention. Accordingly, the scope of protection is not limited by the description set out above. Each and every claim is incorporated into the specification as an embodiment of the present invention.